User Process Checkpoint/Restart

Tera Computer Company
2815 Eastlake Ave E
Seattle, WA 98102

October 9, 1995

1 Introduction

This document describes the design of the Tera user process checkpoint facility. Checkpoint is a means for saving the state of an executing process or group of processes and restarting them later on demand. The motivation for providing checkpoint on the Tera is to allow long running computationally intensive jobs to be restarted after scheduled system down-time or unanticipated system faults. Here are some of the ways that we envision the checkpoint facility might be used:

- application directed
- user command
- automatic checkpoint and recovery
- automated periodic checkpointing

Application directed recovery is provided by system calls embedded in the user application program. User level utilities will be provided to enable users or the system administrator to manually checkpoint a process or group of processes (interface to be designed).

Automatic recovery of batch jobs across scheduled system shutdowns could be provided via an NQS daemon process running as root; of course, this requires integration with NQS. Automatic checkpoint when the system crashes is more problematic; since the system may be in a corrupted state, it might not be possible to checkpoint user programs. To protect against unanticapated system failures, the user may want to periodically checkpoint a job. One way of doing this is via the Unix signal `SIGALRM`, which periodically sends a signal to the process. The user program would need to register a signal handler for `SIGALRM` that calls the `ckpt_create` system call.

2 Interface

An important goal of the design is to provide transparent checkpoint and recovery; it should not be necessary to modify the user program in order for it to be checkpointable. Any user level support that is required should be provided by the standard user runtime that is linked with every user executable by default.

The system call interface is based on POSIX P1003.1a/Draft 12, January, 1995. Note that this is a preliminary draft and has not yet been approved for inclusion in the POSIX standard. Where the POSIX specification is vague regarding state restoration policies, we will attempt to follow the Cray implementation, to ease the conversion process for our customers. The four system calls are described in detail in the subsections below.

2.1 ckpt_setup

This system call sets checkpoint attributes which control the behavior of the `ckpt_create` system call. The attributes are applied to all future checkpoints of the calling process.

```
#include <ckpt.h>
int ckpt_setup(struct ckpt_args *args[],
               size_t nargs);
```

Parameter descriptions:

- `args`: array of pointers to checkpoint attribute structures
- `nargs`: the number of elements in array `args`.

Types of attributes:

- `CKPT_ID`: If the system is unable to restore the process ID or pgrp ID any checkpointed process, the checkpoint and restart shall fail.
- `CKPT_LOCK`: If the system is unable to restore the process ID or pgrp ID any checkpointed process, the checkpoint and restart shall fail.
- `CKPT_SIGNAL`: all checkpointed processes shall receive the specified signal number before resuming execution via restart.

Implementation sketch: Store the attribute list in the proc structure of the calling process. Look up the attributes in `ckpt_create` and store a copy in the checkpoint file. In `ckpt_restart`, read in the attributes from the file and reattach to the proc structure.

2.2 ckpt_create

This system call initiates a checkpoint of a process or group of processes.

```
int ckpt_create(const char *path,
                pid_t id,
                unsigned long type,
                struct ckpt_args *args[],
                size_t nargs);
```

Parameter descriptions:

- `path`: checkpoint file name
- `id`: specifies either a process id, pgrp id, or session id, depending on the value of `type`. If `type == CKPT_PROCESS` and `id == 0`, the calling process is checkpointed.
- `type`: specifies whether to checkpoint a single process, a process group, or a session.
- `args`: checkpoint attributes Question: do these override attributes specified via `ckpt_setup`? (not specified in POSIX) For now, assume these override.
- `nargs`: the number of elements in array `args`.

The call fails if any of the target processes cannot be checkpointed for any reason (see the man page for specific error codes). In order for the calling process to have permission to checkpoint a process, the real or effective user ID must match the id of the subject process, or the calling process has appropriate privileges (e.g. super user).

2.3 ckpt_restart

This system call restarts the process or processes defined by the checkpoint file specified via `path`. POSIX specifies that there may be dependencies on the system state that make it impossible to restore the processes, in which case the restart fails and returns an error code (see the man pages for details on the error codes).

```
pid_t ckpt_restart(const char *path);
```

2.4 ckpt_remove

POSIX does not specify the checkpoint file implementation. It leaves open the possibility that when a checkpoint file is created, some residue of the checkpointed process may be left in the system. The `ckpt_remove` system call provides a means for the system to remove any remaining system residue of the checkpointed process when the checkpoint file is removed.

```
int ckpt_remove(const char *path);
```

The following list summarizes resources that are considered to be unrecoverable. A process owning such resources cannot be checkpointed.

- Processes with open pipes connecting to processes outside the set of processes being checkpointed.
- Processes with open socket connections.

The following will result in restart failure:

- open file has been removed
- lock file is not available

- original executable file has been removed
- file access denied (non-POSIX)

The current design imposes the following additional limitations:

- Shared data is not supported.
- The checkpoint must be restarted on the same OS release. Otherwise shared OS data/program memory may change. This would not matter if restart did not depend on old pointers/stack. (This may not be relevant if we can expunge the checkpoint file of all system state.)
- The system clock will have changed at restart time, disrupting any timings that were in progress when the checkpoint was taken.

3 Checkpoint File Format

POSIX does not specify the checkpoint file format, leaving it up to the implementation. The Tera implementation uses ELF files for both swap and checkpoint images. The name of the file is specified as an argument to the `ckpt_create` system call. Checkpoint is an all-or-nothing operation; if the checkpoint fails for any reason, the target process is not harmed and continues execution. If an error occurs during checkpoint the partially created checkpoint file is removed, to avoid the possibility of leaving a checkpoint file in an incomplete or indeterminate state. Thus the system can assume that if a checkpoint file exists, the checkpoint was completed and all of the necessary information to restore the process is contained in the file.

The checkpoint file contains various system internal state information, and must be protected against user modification, to ensure that the integrity of the operating system is not compromised by a corrupted or deliberately modified checkpoint file. POSIX specifies that no restrictions may be placed on user access to the checkpoint file, although the implementation is permitted to detect modification of the file and deny restart. However, there is no easy way of detecting modification, unless the system maintains a record of all checkpoint files when they are created, which leaves a system residue that must be garbage collected at some point. Also, it is not clear what the benefits are of allowing the user full access to the file if any modifications will result in denial of restart.

An alternative (the approach taken in the old POSIX interface and in the Cray implementation) is to allow renaming of the file, but disallow modifications of the file. One way of doing this is to protect the file via the file permissions mode together with a special file attribute bit. For instance, file mode `0400` allows read-only access by the owner; setuid processes are protected against user read access via checkpoint file permission mode `000`. The special file attribute bit prevents the owner from changing the file permissions mode (i.e. prohibits `chown(2)` and `chmod(2)`); kernel modifications are necessary to implement the file attribute bit and to enforce the restrictions on file access.

4 Summary of Recoverable Resources

Various resources held by the process or group of processes when the checkpoint occurs must be restored when the process is restarted. This includes open file state, memory state, and various

process state. In addition, the checkpoint must handle any system calls that are blocked while waiting for I/O when the checkpoint occurs. These requirements are discussed in detail in the subsections below.

4.1 Open File State

Saving and restoring of open file state is typically the most complex part of checkpoint implementations. The current design attempts to push the complexity up to the user level, to minimize the kernel internal modifications. A user level implementation is easier to debug, maintain, and extend.

Each process has an open file table that is allocated in user memory. The user level wrappers for various I/O related system calls record information about the operation in the table. When a process is checkpointed, the system sends the process a SIGCKPT signal. The user signal handler for `SIGCKPT` updates the information for each entry in the table by calling `stat(2)` and `lseek(2)` to obtain the current file offset pointer. When the checkpoint is restarted, the system sends the restarted process a `SIGCKPTRESTART`. The user signal handler for `SIGCKPTRESTART` is responsible for restoring the open file state as recorded in the table. The table contents are in user memory, thus are saved and restored automatically. When a process calls `execve(2)`, the file state information is automatically transferred to the new memory image.

For each open file, the following information is saved and restored.

- file path
- file descriptor number
- file offset
- flags given to open
- access mode
- fstat information (not restored; used only for verification)

In addition to the open file state, the following information is saved and restored.

- current working directory
- root directory
- file creation mask

The root directory is the starting point for path searches of pathnames beginning with "/". Only the superuser is allowed to change the root directory of a process (via chroot(2)). When the root directory is changed, all the saved file paths become invalid. So, if a process opens a file then changes its root directory, the restart will fail because the saved file path is no longer valid.

When a file is open by some process, but has been removed, the file is considered to be unlinked. An unlinked file is marked for destruction when the last process that has the file open closes the file or exits. A fairly common use of unlinked files is temporary files (i.e. files residing on `/tmp`). Unlinked files require special treatment for checkpoint/restart, since the file could disappear when

the original process completes execution and closes the file. A subsequent restart would then fail, since the file would no longer exist. A possibility is to provide a user option for saving a copy of all unlinked files as part of the checkpoint.

Another potential problem occurs when a process opens a file, then renames it. The renaming invalidates the original path, so the restart will fail. This is similar to the case where the file is renamed after the checkpoint completes. When the checkpoint is restarted, the original path is no longer valid.

The `mmap` and `mmap_fsblk` system calls are another area of difficulty.

In addition to regular files, file locks, terminal, and network connections must also be saved and restored:

- file locks: reacquire the lock (Tera OS does not support mandatory locks, so this only applies to advisory locks).
- controlling tty: if the checkpointed process had a ctty, the restarted process inherits the ctty of the process that called `ckpt_restart`.
- pseudo ttys: save/restore data in the tty queue
- pipes: save/restore data in the pipe
- sockets: not recoverable
- NFS files: do these require any special support?

4.2 Blocked System Calls

When a checkpoint is requested, the target process may have multiple concurrent system calls executing, some of which may be blocked within the OS. The checkpoint cannot simply take a snapshot of the blocked calls, since the calls may be holding system resources, such as locks, that will not be available at restart. Thus, all system calls must either be allowed to complete, or be interrupted and reissued after the checkpoint.

A system call can be blocked interruptibly or non-interruptibly. Non-interruptible waits are for atomic operations, such as disk I/O. This type of wait is implemented via sleep() or suspend locks. The checkpoint will not occur until all such calls are unblocked and complete. Happily, this is already ensured by the signal delivery mechanisms, and checkpoint is initiated via a signal (SIGCKPT), so nothing special needs to be done.

[TODOSC: currently, there is no mechanism in place for resending the signal when the SPChore becomes unblocked; Rich will add this support.]

Interruptible waits are usually due to slow I/O (e.g. reading from a tty), or waiting for an event to occur (e.g. sigpause(2)). This type of wait is implemented as a call to tsleep with PCATCH set. These calls are interrupted and transparently restarted if possible. Note that system calls that have already committed cannot be restarted, but instead return a partial success (e.g. a short read count) or an error code. The BSD4.4 code already takes care of this case (examines the error code returned by tsleep and converts it if necessary).

In order for calls to be restarted, the user process must have called sigaction(2), specifying `SA_RESTART` for the `SIGCKPT` signal (should be done automatically by user runtime when registering the `SIGCKPT`

handler). The following system calls may be restarted: wait, ioctl, read, write, sendto, recvfrom, sendmsg and recvmsg on a communications channel or a slow device (e.g tty, but not a regular file).

The implementation leverages the signal delivery code to interrupt all SPChores that are blocked interruptibly and return ERESTART. When the unblocked SPChore returns to the supervisor call entry wrapper, the the error code is converted to ECKPTAGAIN and the chore returns to the user level system call wrapper. The OS maintains an internal flag that indicates checkpoint is in progress, so all system calls are caught in the OS `svc_entry` wrapper and return `ECKPTAGAIN`. The chore loops in the user wrapper, retrying the system call, until domain signal is set when suspending the task in preparation for checkpoint. The user chore state is saved in the checkpoint file as part of the user memory image, and the system call is automatically restarted when the checkpoint is restarted. Blocking in user mode has the benefit of reducing the amount of system state that must be stored in the checkpoint file.

```
user_foo() {
    do {
        result = system_foo();
    } while (errno == ECKPTAGAIN);
    return result;
}
```

CAUTION: this requires that user chores must not have domain signal masked when calling the user system call stub. Otherwise, the team will not respond to domain signal, and the system will eventually kill the task. There are a few exceptions, notably the system calls used for handshaking between the user runtime and OS during pm-swap (`task_swapsave_complete`, `team_swapsave_complete`, `tera_return_stream`). The OS never returns `ECKPTAGAIN` for these calls.

Note: we rely on the user runtime to ensure that the task is quiescent, and that no system calls are executing (i.e. all outstanding system calls are blocked) when `tera_signal_number` is invoked. If the user runtime neglects to ensure this condition, the OS is still safe, since this will not result in inconsistencies in the system state. However, the restarted process will likely not work correctly, and this could be very difficult to debug. TODO: consider adding a mechanism to verify that no calls are in progress, and deny checkpoint otherwise.

4.3 Memory

The memory image of a checkpointed task is saved in the same format as the swap image. This allows the same mechanisms to be used when saving and reloading the memory image for both checkpoint/restart and swapping. The `swapin` routine simply skips over ELF segments that are specific to checkpoint files (such as the segments holding the saved process and open file state information).

The program text is not saved in the image file. It is reloaded from the original executable when the task is swapped into memory. When swapping a task out of memory, the file is marked in the file system as busy; this prevents the executable from being removed prior to swapin. Unfortunately, this cannot be done in the case of checkpoint, since there is no way to know when the busy flag can be safely removed. Instead, either the executable must be saved as part of the checkpoint, or sufficient information must be saved describing the executable to enable any modification of the

executable to be detected at restart. If restart detects modification of the file, restart is denied. Restart will also fail if the executable references shared libraries, and the required library version is not available. Shared library data is currently task-private data, thus is saved as part of the task data memory image.

The POSIX draft specifies the following memory state shall be restored for restarted processes, otherwise the restart shall fail.

- process memory image: saved via swapin/swapout mechanisms.
- locked memory regions: handle in `DataAddressSpace` restoration
- mapped memory regions: handled in the same manner as as for swapping.
- protected memory regions: handle in `DataAddressSpace` restoration
- shared memory regions: not applicable, since the Tera OS does not currently support System V style shared segments.

4.4 Process State

The POSIX draft specifies that the following Unix process attributes shall be saved and restored.

- scheduling policy and priority
- real and effective user and group IDs and supplementary group IDs.
- file mode creation mask
- current working directory and root directory
- parent/child relationships between the restarted processes (only relevant when checkpointing a process group or session)
- process signal state
  - pending signals
  - process signal mask (blocked and/or ignored signals)
  - signal handlers: these are automatically saved as part of the user runtime memory image, thus nothing special is required.
- process timer state, as if no time had passed since the process was checkpointed.
- message queue descriptors: not applicable since messages are not supported.
- thread attributes and execution state: these are saved as usual by the user runtime during pm-swap, so nothing special is required.
- usage statistics (for accurate billing), specifically: `tms_utime, tms_stime, tms_cutime, tms_cstime` (user CPU time, system CPU time, user CPU time of terminated child processes, system CPU time of terminated child processes)

- controlling terminal: set to ctty of the process that called `ckpt_restart`. If the calling process does not have a ctty and the checkpointed process did, the restart shall fail.

In addition to the Unix process state, information adequate to reconstruct the microkernel `TaskControlBlock`, `TeamControlBlock`, `SPTask`, and `SPTeam` structures is saved.

- taskid
- parent taskid (not required to be recoverable)
- team attributes (for creating new teams)
- priority
- nice
- number of teams
- resource usage information
- per-team protection domain info (e.g. event and stream counters)

The `ckpt_setup` system call provides a means for the user to specify whether the process id and/or lock file must be recovered. The default action is to return `EBUSY` to the `ckpt_restart` call if a resource resource associated with the target process is in use by the system and thus not available (for instance, the process id). An exception is the parent process id; this allows orphaned processes to be restarted.

If this was a checkpoint of a process group, the process group id must be restored; likewise for sessions. POSIX does not specify whether the process group id and session membership should be saved and restored for a single process checkpoint. The Tera implementation will put the restarted process in the process group and session of the caller of `ckpt_restart`; the caller will also be the parent of the process.

In addition to the POSIX requirements, it may be useful to save and restore the following state:

- NQS resource quota limits
- NQS resources consumed

4.5 Processor Context

The only processor context that needs to be saved and restored is the protection domain state registers, such as the event and stream counters. This state is normally saved during pm-swap into the `TeamControlBlock` structures. The register state of all executing user streams is saved by the user runtime into the user private data area; this memory is saved and restored automatically. Likewise, the register state of all SPChores is saved by the supervisor runtime into the supervisor task-private data memory. So nothing special needs to be done.

5 Implementation Sketch

Checkpoint implementations can be fairly complex, mainly due to complications in the recovery of the system state associated with a process. The Tera implementation leverages existing code paths as much as possible, minimizing the amount of dedicated code that must be maintained. The user level checkpoint and recovery routines are invoked via Unix signals. The usual pm-swap mechanism is used to quiesce the user process prior to checkpoint, and the memory swap mechanisms are used to write out and read in the memory image.

5.1 Checkpoint Mechanisms

The `ckpt_create` system call verifies the caller has the appropriate permissions, then generates a `SIGCKPT` for the process to be checkpointed. When the signal handler has completed execution, control is returned to the kernel, which then suspends the task and saves the task memory image along with additional system state sufficient to restart the task. It is important that no other file operations execute after the state has been saved, to prevent the saved state from becoming stale. We accomplish this by transferring control directly to the system when the `SIGCKPT` signal handler completes execution.

Brief summary of Tera signal handling mechanisms: The signal delivery code arranges to set domain signal in all the user domains, emulating a pm-swap. The user saves its state, and the last user stream calls `task_swapsave_complete()` to notify the OS. When `task_swapsave_complete` returns, one stream on each team is created. One of those streams goes on to execute `tera_signal_number()` while the others spin, waiting for direction from that stream (to say that signal processing has completed). The signal handling stream repeatedly calls `tera_signal_number` to obtain the pending signal number, and invokes the handler, until all signals have been handled.

If there are multiple signals pending, the kernel ensures that the `SIGCKPT` signal is the first signal to be returned to the user by `tera_signal_number`. When the signal handler completes execution and calls `tera_signal_number`, control is returned to the kernel, which initiates checkpointing of the task. Note that any additional pending signals are not delivered until after the checkpoint has completed.

5.2 Restart Mechanisms

The Unix signal mechanism is used to invoke a user level recovery routine when the checkpointed process is restarted.

- OS constructs the task and restores internal state.
- OS arranges to have a stream that is blocked in `tera_task_swapsave_complete()` return to the user.
- The user stream calls `tera_signal_number()`, which returns `SIGCKPTRESTART` as the first signal to handle.
- The user handler for `SIGCKPTRESTART` calls the user level recovery routine (e.g. restore open file state).

The `ckpt_setup` and `ckpt_create` system calls have an option for the user to specify a signal to be sent to a restarted process. Note that this signal is distinct from `SICKPTGRESTART`, which is used explicitly for handshaking between the Tera OS and the Tera user runtime.

5.3 Staged Implementation

The first stage will focus on implementing the basic mechanisms to initiate a checkpoint, save the process image, and restart it. In the second stage support will be provided for the more complicated issues such as blocked system calls and recovery of I/O state.

- define the system call interface
- define the checkpoint file format
- define mechanisms to save/restore system state
- test, debug checkpoint/restart of a single process
- save/restore open file state
- handle blocked system calls
- support for checkpointing process groups and sessions (TODO)

Once a complete implementation exists, additional optimizations and/or extensions can be considered. For instance, these are some of the common optimizations implemented by existing systems to reduce the checkpoint overhead. Note however, that these solutions are not necessarily suitable for use on the Tera, so other solutions may need to be devised.

- user directed checkpointing: save only "inuse" memory (This seems like a reasonable optimization on the Tera.)
- incremental checkpoint: via copy-on-write (copy-on-write is not currently supported in the Tera OS)
- asynchronous checkpoint: main memory checkpointing via fork (without copy-on-write, this may not help much)